23 | 数字计算：介绍线性代数和数值计算库

2020-01-17 吴咏炜

讲述：吴咏炜
时长 12:31 大小 11.48M

你好，我是吴咏炜。

科学计算在今天已经完全可以使用 C++ 了。我不是从事科学计算这一领域的工作的，不过，在工作中也多多少少接触到了一些计算相关的库。今天，我就给你介绍几个有用的计算库。

Armadillo

说到计算，你可能首先会想到矩阵、矢量这些东西吧？这些计算，确实就是科学计算中的常见内容了。这些领域的标准，即是一些 Fortran 库定下的，如：

BLAS [1]

LAPACK [2]

ARPACK [3]

它们的实现倒不一定用 Fortran，尤其是 BLAS：

OpenBLAS [4] 是用汇编和 C 语言写的

Intel MKL [5] 有针对 Intel 的特定 CPU 指令集进行优化的汇编代码

Mir GLAS [6] 是用 D 语言写的

不管实现的方法是哪一种，暴露出来的函数名字是这个样子的：

ddot

dgemv

dsyrk

sgemm

……

这个接口的唯一好处，应该就是，它是跨语言并且跨实现的😅。所以，使用这些函数时，你可以切换不同的实现，而不需要更改代码。唯一需要修改的，通常就是链接库的名字或位置而已。

假设我们需要做一个简单的矩阵运算，对一个矢量进行旋转：

$$
\begin{aligned}
\mathbf{P} &= \begin{bmatrix} 1 \\ 0 \end{bmatrix} \\
\mathbf{R} &= \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix} \\
\mathbf{P}' &= \mathbf{R} \cdot \mathbf{P}
\end{aligned}
$$

这么一个简单的操作，用纯 C 接口的 BLAS 来表达，有点痛苦：你需要使用的大概是 `dgemv_` 函数，而这个函数需要 11 个参数！我查阅了一下资料之后，也就放弃了给你展示一下如何调用 `dgemv_` 的企图，我们还是老老实实地看一下在现代 C++ 里的写法吧：

复制代码

```cpp
#include <armadillo>
#include <cmath>
#include <iostream>

using namespace std;

int main()
{
  //  代表位置的向量
  arma::vec pos{1.0, 0.0};

  //  旋转矩阵
  auto& pi = arma::datum::pi;
  double angle = pi / 2;
  arma::mat rot = {
    {cos(angle), -sin(angle)},
    {sin(angle), cos(angle)}};

  cout << "Current position:\n"
       << pos;
  cout << "Rotating "
       << angle * 180 / pi
       << " deg\n";

  arma::vec new_pos = rot * pos;
  cout << "New position:\n"
       << new_pos;
}
```

这就是使用 Armadillo [7] 库来实现矢量旋转的代码。这个代码，基本就是上面的数学公式的一一对应了。代码相当直白，我只需要稍稍说明一下：

所有的 Armadillo 的类型和函数都定义在 `arma` 名空间下。

Armadillo 在 `arma::datum` 下定义了包括 pi 和 e 在内的一些数学常量。

`vec` 是矢量类型，`mat` 是矩阵类型，这两个类型实际上是 `Col<double>` 和 `Mat<double>` 的缩写别名。

Armadillo 支持使用 C++11 的列表初始化语法来初始化对象。

Armadillo 支持使用流来输出对象。

上面代码的输出为：

```
Current position:
1.0000
0
Rotating 90 deg
New position:
6.1232e-17
1.0000e+00
```

输出里面的 `6.1232e-17` 是浮点数表示不精确的后果，把它理解成 0 就对了。

我们上面已经提到了 `vec` 实际上是 `Col<double>`，双精度浮点数类型的列矢量。自然，Armadillo 也有行矢量 `rowvec`（即 `Row<double>`），也可以使用其他的数字类型，如 `int`、 `float` 和 `complex<float>`。此外，除了大小不确定的线性代数对象之外，Armadillo 也提供了固定大小的子类型，如 `vec::fixed<2>` 和 `mat::fixed<2, 2>`；为方便使用，还提供了不少别名，如 `imat22` 代表 `Mat<int>::fixed<2, 2>` 等。固定大小的对象不需要动态内存分配，使用上有一定的性能优势。

Armadillo 是一个非常复杂的库，它的头文件数量超过了 500 个。我们今天不可能、也不必要描述它的所有功能，只能稍稍部分列举一下：

除了目前提到的列矢量、行矢量和矩阵外，Armadillo 也支持三维的数据立方体，`Cube` 模板。

Armadillo 支持稀疏矩阵，`SpMat` 模板。

除了数学上的加、减、乘运算，Armadillo 支持按元素的乘法、除法、相等、不等、小于比较等（使用 `%`、`/`、`==`、`!=`、`<` 等）运算，结果的大小跟参数相同，每个元素是相应运算的结果。某些运算符可能不太直观，尤其是 `%`（不是取模）和 `==`（返回不是单个布尔值，而是矩阵）。

Armadillo 支持对非固定大小的矢量、矩阵和立方体，改变其大小（`.reshape()` 和 `resize()`）。

Armadillo 可以方便地按行（`.col()`）、列（`.row()`）、对角线（`.diag()`）读写矩阵的内容，包括用一个矢量去改写矩阵的对角线。

Armadillo 可以方便地对矩阵进行转置（`.t()`）、求反（`.inv()`）。

Armadillo 可以对矩阵进行特征分解（`eigen_sym()`、`eigen_gen()` 等）。

Armadillo 支持傅立叶变换（`fft()`、`fft2()` 等）。

Armadillo 支持常见的统计计算，如平均值、中位值、标准偏差等（`mean()`、`median()`、`stddev()` 等）。

Armadillo 支持多项式方程求根（`roots`）。

Armadillo 支持 k-平均聚类（*k*-means clustering）算法（`kmeans`）。

等等。

如果你需要用到这些功能，你可以自己去查看一下具体的细节，我们这儿只提几个与编程有关的细节。

对象的输出

我们上面已经展示了直接把对象输出到一个流。我们的写法是：

复制代码

```
cout << "Current position:\n"
     << pos;
```

实际上基本等价于调用 `print` 成员函数：

复制代码

```
pos.print("Current position:");
```

这个写法可能会更简单些。此外，在这两种情况，输出的格式都是 Armadillo 自动控制的。如果你希望自己控制的话，可以使用 `raw_print` 成员函数。比如，对于上面代码里对 `new_pos` 的输出，我们可以写成（需要包含 <iomanip>）：

复制代码

```
cout << fixed << setw(9)
     << setprecision(4);
new_pos.raw_print(
  cout, "New position:");
```

这种情况下，你可以有效地对格式、宽度和精度进行设置，能得到：

```
New position:
0.0000
1.0000
```

记得我们说过 `vec` 是 `Col<double>` 的别名，因此输出是多行的。我们要输出成单行的话，转置（transpose）一下就可以了：

复制代码

```
cout << fixed << setw(9)
     << setprecision(4);
new_pos.t().raw_print(
  cout, "New position:");
```

输出为：

```
New position:
0.0000 1.0000
```

表达式模板

如果你奇怪前面 `dgemv_` 为什么有 11 个参数，这里有个我没有提的细节是，它执行的实际上是个复合操作：

$$\mathbf{y} \leftarrow \alpha \mathbf{A} \cdot \mathbf{x} + \beta \mathbf{y}$$

如果你只是简单地做乘法的话，就相当于 α 为 1、β 为 0 的特殊情况。那么问题来了，如果你真的写了类似于上面这样的公式的话，编译器和线性代数库能不能转成合适的调用、而没有额外的开销呢？

答案是，至少在某些情况下是可以的。秘诀就是表达式模板（expression template）[8]。

那什么是表达式模板呢？我们先回过去看我上面的例子。有没有注意到我写的是：

```
arma::vec new_pos = rot * pos;
```

而没有使用 `auto` 来声明？

其中部分的原因是，`rot * pos` 的类型并不是 `vec`，而是：

```
const Glue<Mat<double>, Col<double>, glue_times>
```

换句话说，结果是一个表达式，而并没有实际进行计算。如果我用 `auto` 的话，行为上似乎一切都正常，但我每次输出这个结果时，都会重新进行一次矩阵的乘法！而我用 `arma::vec` 接收的话，构造时就直接进行了计算，存储了表达式的结果。

上面的简单例子不能实际触发对 `dgemv_` 的调用，我用下面的代码实际验证出了表达式模板产生的优化（`fill::randu` 表示对矢量和矩阵的内容进行随机填充）：

```
#include <armadillo>
#include <iostream>

using namespace std;
using namespace arma;


```

```
int main()
{
  vec x(8, fill::randu);
  mat r(8, 8, fill::randu);
  vec result = 2.5 * r * x;
  cout << result;
}
```

赋值语句右边的类型是：

```
const Glue<eOp<Mat<double>,
                eop_scalar_times>,
           Col<double>, glue_times>
```

当使用这个表达式构造 `vec` 时，就会实际发生对 `dgemv_` 的调用。我也确实跟踪到了，在将要调用 `dgemv_` 时，标量值 2.5 确实在参数 `alpha` 指向的位置上（这个接口的参数都是指针）。

从上面的描述可以看到，表达式模板是把双刃剑：既可以提高代码的性能，又能增加代码被误用的可能性。在可能用到表达式模板的地方，你需要注意这些问题。

平台细节

Armadillo 的文档里说明了如何从源代码进行安装，但在 Linux 和 macOS 下通过包管理器安装可能是更快的方式。在 CentOS 下可使用 `sudo yum install armadillo-devel`，在 macOS 下可使用 `brew install armadillo`。使用包管理器一般也会同时安装常见的依赖软件，如 ARPACK 和 OpenBLAS。

在 Windows 上，Armadillo 的安装包里自带了一个基本版本的 64 位 BLAS 和 LAPACK 库。如果需要更高性能或 32 位版本的话，就需要自己另外去安装了。除非你只是做一些非常简单的线性代数计算（就像我今天的例子），那直接告诉 Armadillo 不要使用第三方库也行。

> `cl /EHsc /DARMA_DONT_USE_BLAS /DARMA_DONT_USE_LAPACK ...`

Boost.Multiprecision

众所周知，C 和 C++（甚至推而广之到大部分的常用编程语言）里的数值类型是有精度限制的。比如，上一讲的代码里我们就用到了 `INT_MIN`，最小的整数。很多情况下，使用目前这些类型是够用的（最高一般是 64 位整数和 80 位浮点数）。但也有很多情况，这些标准的类型远远不能满足需要。这时你就需要一个高精度的数值类型了。

有一次我需要找一个高精度整数类型和计算库，最后找到的就是 Boost.Multiprecision [9]。它基本满足我的需求，以及一般意义上对库的期望：

- 正确实现我需要的功能
- 接口符合直觉、易用
- 有良好的性能

正确实现功能这点我就不多讲了。这是一个基本出发点，没有太多可讨论的地方。在我上次的需求里，对性能其实也没有很高的要求。让我对 Boost.Multiprecision 满意的主要原因，就是它的接口了。

接口易用性

我在 [第 12 讲] 提到了 CLN。它对我来讲就是个反面教材。它的整数类型不仅不提供 `%` 运算符，居然还不提供 `/` 运算符！它强迫用户在下面两个方案中做出选择：

- 使用 `truncate2` 函数，得到一个商数和余数
- 使用 `exquo` 函数，当且仅当可以整除的时候

不管作者的设计原则是什么，这简直就是易用性方面的灾难了——不仅这些函数要查文档才能知晓，而且有的地方我真的只需要简单的除法呀……

哦，对了，它在 Windows 编译还很不方便，而我那时用的正是 Windows。

Boost.Multiprecision 的情况则恰恰相反，让我当即大为满意：

使用基本的 `cpp_int` 对象不需要预先编译库，只需要 Boost 的头文件和一个好的编译器。

常用运算符 `+`、`-`、`*`、`/`、`%` 一个不缺，全部都有。

可以自然地通过整数和字符串来进行构造。

提供了用户自定义字面量来高效地进行初始化。

在使用 IO 流时，输入输出既可以使用十进制，也可以通过 `hex` 来切换到十六进制。

下面的代码展示了它的基本功能：

复制代码

```
#include <iomanip>
#include <iostream>
#include <boost/multiprecision/cpp_int.hpp>

using namespace std;

int main()
{
  using namespace boost::
    multiprecision::literals;
  using boost::multiprecision::
    cpp_int;

  cpp_int a =
    0x123456789abcdef0_cppi;
  cpp_int b = 16;
  cpp_int c{"0400"};
  cpp_int result = a * b / c;
  cout << hex << result << endl;
  cout << dec << result << endl;
}
```

输出是：

```
123456789abcdef
81985529216486895
```

我们可以看到，`cpp_int` 可以通过自定义字面量（后缀 `_cppi`；只能十六进制）来初始化，可以通过一个普通整数来初始化，也可以通过字符串来初始化（并可以使用 `0x` 和 `0` 前缀来选择十六进制和八进制）。拿它可以正常地进行加减乘除操作，也可以通过 IO 流来输入输出。

性能

Boost.Multiprecision 使用了表达式模板和 C++11 的移动来避免不必要的拷贝。后者当然是件好事，而前者曾经坑了我一下——我第一次使用 Boost.Multiprecision 时非常困惑为什么我使用 `half(n - 1)` 调用下面的简单函数居然会编译不过：

复制代码

```
template <typename N>
inline N half(N n)
{
  return n / 2;
}
```

我的意图当然是 `N` 应当被推导为 `cpp_int`，`half` 的结果也是 `cpp_int`。可实际上，`n - 1` 的结果跟上面的 Armadillo 展示的情况类似，是另外一个单独的类型。我需要把 `half(n - 1)` 改写成 `half(N(n - 1))` 才能得到期望的结果。

我做的计算挺简单，并不觉得表达式模板对我的计算有啥帮助，所以我最后是禁用了表达式模板：

复制代码

```
typedef boost::multiprecision::
  number<
    boost::multiprecision::
      cpp_int_backend<>,
    boost::multiprecision::et_off>
    int_type;
```

类似于 Armadillo 可以换不同的 BLAS 和 LAPACK 实现，Boost.Multiprecision 也可以改换不同的后端。比如，如果我们打算使用 GMP [10] 的话，我们需要包含利用 GMP 的头文件，并把上面的 `int_type` 的定义修正一下：

```cpp
#include <boost/multiprecision/gmp.hpp>

typedef boost::multiprecision::
  number<
    boost::multiprecision::gmp_int,
    boost::multiprecision::et_off>
    int_type;
```

注意，我并不是推荐你换用 GMP。如果你真的对性能非常渴求的话，应当进行测试来选择合适的后端。否则缺省的后端易用性最好——比如，使用 GMP 后端就不能使用自定义字面量了。

我当时寻找高精度算术库是为了做 RSA 加解密。计算本身不复杂，属于编程几小时、运行几毫秒的情况。如果你有兴趣的话，可以看一下我那时的挑选过程和最终代码 [11]。

Boost 里好东西很多，远远不止这一样。下一讲我们就来专门聊聊 Boost。

内容小结

本讲我们讨论了两个进行计算的模板库，Armadillo 和 Boost.Multiprecision，并讨论了它们用到的表达式模板技巧和相关的计算库，如 BLAS、LAPACK 和 GMP。可以看到，使用 C++ 你可以站到巨人肩上，轻松写出高性能的计算代码。

课后思考

性能和易用性往往是有矛盾的。你对性能和易用性有什么样的偏好呢？欢迎留言与我分享。

参考资料

[1] Wikipedia, “Basic Linear Algebra Subprograms”. https://en.wikipedia.org/wiki/Basic_Linear_Algebra_Subprograms

[2] Wikipedia, “LAPACK”. https://en.wikipedia.org/wiki/LAPACK

[3] Wikipedia, “ARPACK”. https://en.wikipedia.org/wiki/ARPACK

[4] Zhang Xianyi et al., OpenBLAS. https://github.com/xianyi/OpenBLAS

[5] Intel, Math Kernel Library. https://software.intel.com/mkl

[6] Ilya Yaroshenko, mir-glas. https://github.com/libmir/mir-glas

[7] Conrad Sanderson and Ryan Curtin, “Armadillo: C++ library for linear algebra & scientific computing”. http://arma.sourceforge.net/

[8] Wikipedia, “Expression templates”. https://en.wikipedia.org/wiki/Expression_templates

[9] John Maddock, Boost.Multiprecision. https://www.boost.org/doc/libs/release/libs/multiprecision/doc/html/index.html

[10] The GNU MP bignum library. https://gmplib.org/

[11] 吴咏炜, “Choosing a multi-precision library for C++—a critique”. https://yongweiwu.wordpress.com/2016/06/04/choosing-a-multi-precision-library-for-c-a-critique/

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇　22 | 处理数据类型变化和错误：optional、variant、expected和Herbception

下一篇　24 | Boost：你需要的“瑞士军刀”

精选留言 (4)

写留言

廖熊猫
2020-01-17

易用的东西灵活性就稍微弱一些，在能快速完成业务的时候，我还是比较偏向易用性，一查文档马上就可以用起来，真的出现性能瓶颈的时候再去折腾复杂性能好的东西。
给需要学习线性代数的小伙伴推荐一个教材：http://textbooks.math.gatech.edu/ila/index.html
需要点英语基础才行。

展开

作者回复: 我的观点和你差不多……😂

那个在线教材真棒！谢谢推荐。

2

秀
2020-02-01

我发现留言的都是大佬。学习了学习了。

展开

花晨少年
2020-01-18

科学计算这块最近几年突然热起来了，在调研xdl深度学习框架时，推理引擎cpu版本用的就是mkl库做各种矩阵运算，但是可能还是gpu版本的cuda库应用更广泛一些，听说cuda编程挺难的，需要了解很多异构硬件的细节。

展开

作者回复: 如何用 C++ 做异构计算，这是一个新领域。可以留意一下 SYCL。

三味
2020-01-17

当然更偏向易用性。实现更重要。一个算法不确定是否能实现，先从易用的库开始快速迭代算法实现。优化是最后要考虑的。
还有就是在选择第三方库的时候，我倾向于选择纯头文件的库。比如Eigen3。我用Eigen3无非就是矩阵计算，能够快速求解线性方程组的解就好，并没有高次方程求根这种（其实也有，为了求一个三次方程就要引用一个库，我选择找一个现成的实现）。...

展开

作者回复: 看来你是真正搞计算的啊……我是偶尔碰一下而已。

《数学与泛型编程》绝对是好书。这本已经算是作者的另一本书的简化版本了（那本书更抽象，我也只啃了个开头而已了）。对于喜欢数学的程序员，我绝对大力推荐。